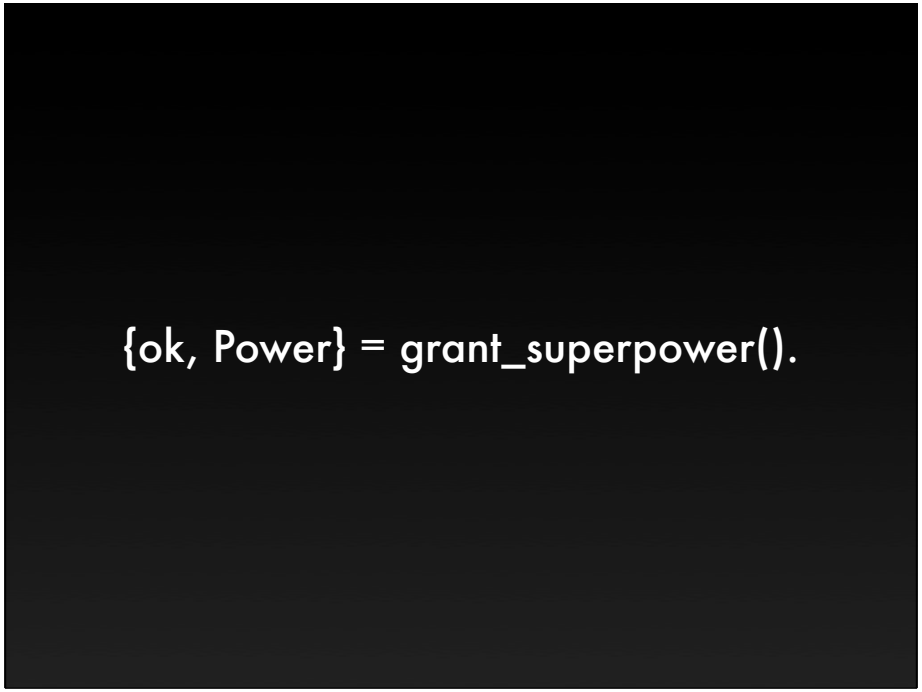
Erlang Gives You Superpowers!

Jack Moffitt
jack@metajack.im
@metajack
http://metajack.im

* unlike most super powers, no spider bites, radiation exposure, family misfortune, or alien birth required
* like any other super thing, there are weaknesses

Pattern Matching


{ok, Power} = grant_superpower().

```
handle_call({submit_word, Word},
            {Player, _},
            State = #game{
               words=Words,
               scores=[Score1, Score2],
               player_data=[
               #player{rating=Rating1} = PlayerData1,
               #player{rating=Rating2} = PlayerData2]}) ->

    %% ...

    {reply, ok, NewState}.
```


Binary Syntax

Verifies *and* binds simultaneously.

MP3 Frame Header

```
decode_header(<<2#11111111111:11,B:2,C:2,_D:1,E:4,F:2,G:1,Bits:9>>) ->

    %% header code
```

Compared to other languages, they are effectively free.

Completely removes the need for non-blocking programming style like Twisted Python, Node.js, and C select/poll loops.

In Joe's book, 20,000 took 10 microseconds of wall time. so did 50,000. So did 200,000.

Easy to reason about.

Selective receive is also very powerful. For instance you can spawn a ton of processes to do something to a list, given them a index. They reply with the index and the value. The events all arrive out of order when they are done, but you receive them in order and get a sorted version.

PIDs are for the whole cluster. All the normal stuff you do in Erlang is exactly the same for the distributed case, except the latency is higher.


OTP

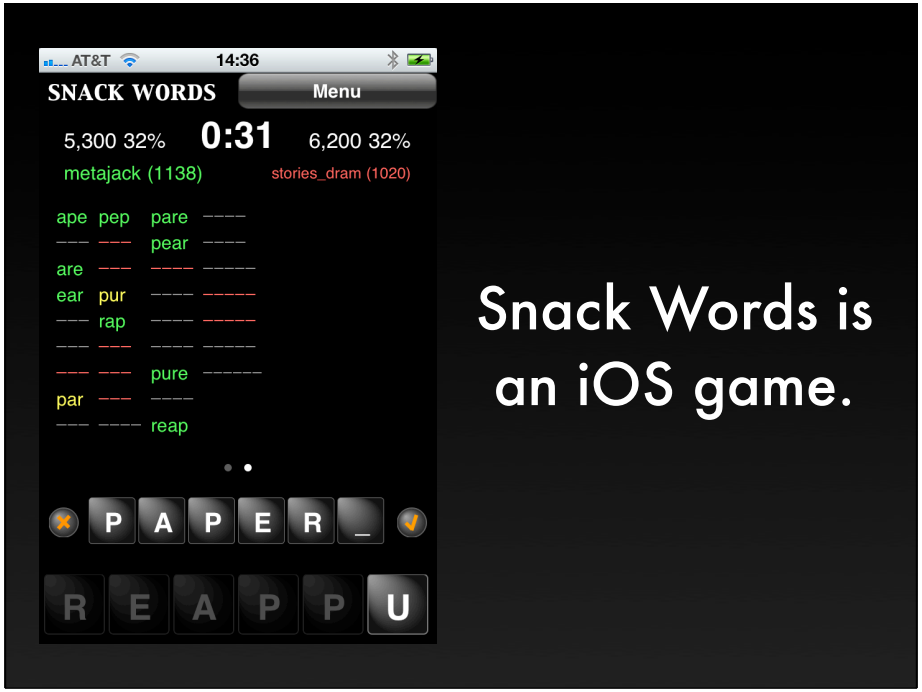
Tooling

Dialyzer, QuickCheck, rebar, Tidier

Apps and Libraries

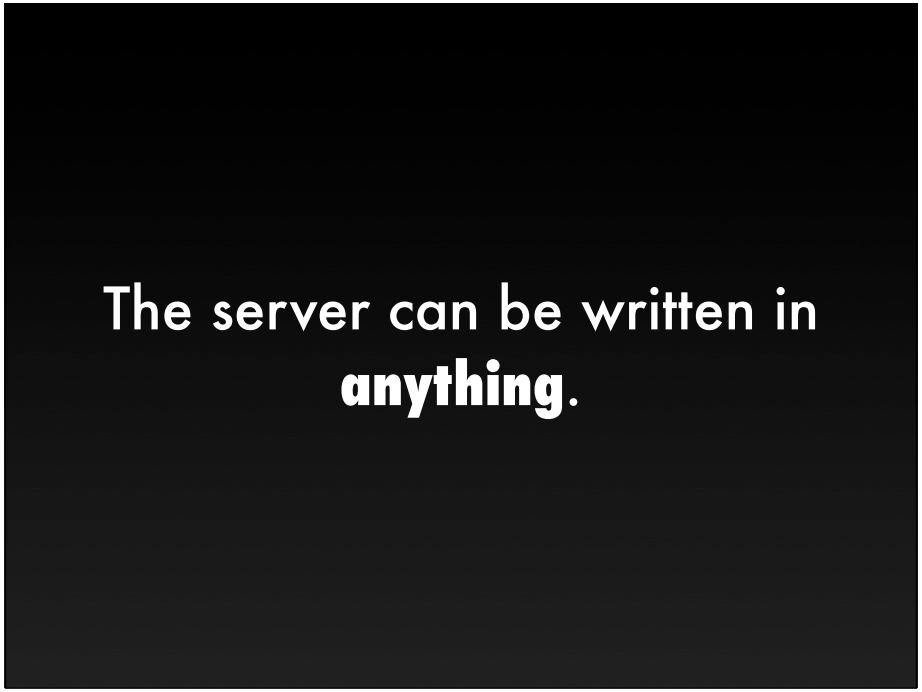
ejabberd, Riak, CouchDB, webmachine, RabbitMQ


Mild-mannered
Erlang



Snack Words is an iOS game.

It's not my favorite, but it certainly is the easiest choice.



This goes both ways. I could use the coolest thing, but also, the user's will never care as long as it works.


The goal is to
keep it simple
and write it
fast.


What to use?

Use **Erlang** of course.

But wait!
It also needs a protocol.

What's cheap to implement on iOS?


The Protocol

Length prefixed **JSON** packets.

gen_tcp provides

{packet, N}

for N = 1,2,4

Receive:
{tcp, Socket, Data}

Decode:
mochijson2:decode(Data)

Encode:

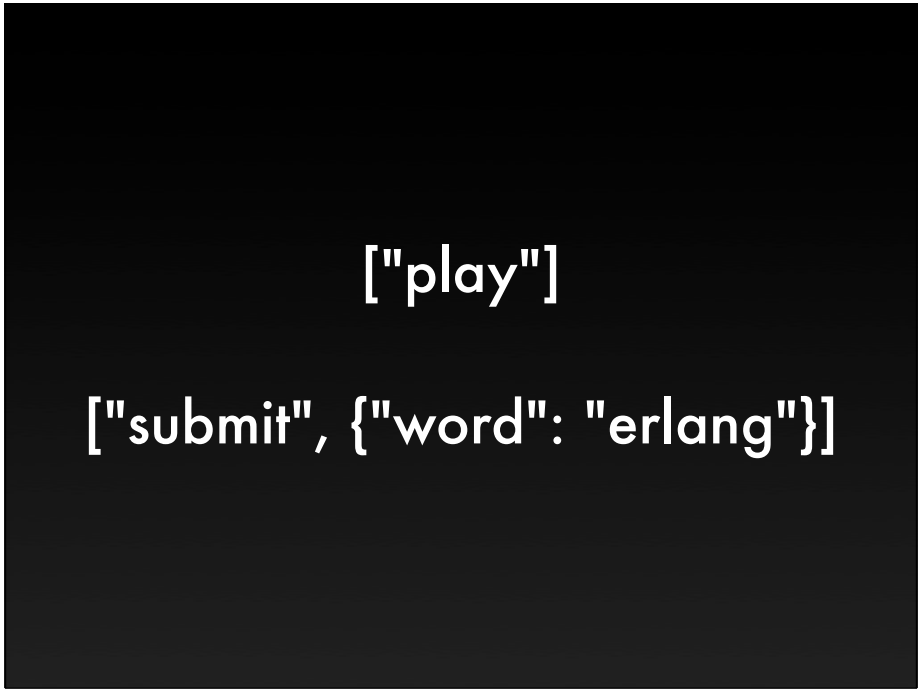
Data = mochijson2:encode(Json)

Send:

gen_tcp:send(Socket, Data)

Objective-C:

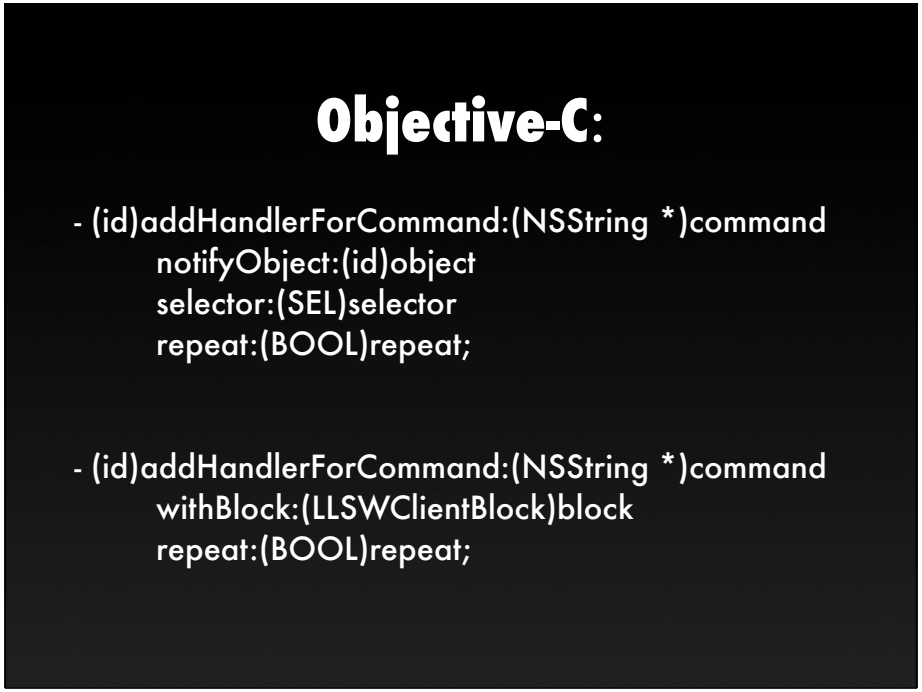
```
-	(void)sendAvailableData {
		NSUInteger sentBytes, maxLen;
		do {
			if (outBuffer == nil) {
				if ([sendQueue count] > 0) {
					id jsonObj = [sendQueue objectAtIndex:0];
					[sendQueue removeObjectAtIndex:0];
					NSString *jsonString = [jsonObj JSONRepresentation];

					NSLog(@"SENDING: %@", jsonString);

					NSData *data = [jsonString dataUsingEncoding:NSUTF8StringEncoding];
					NSUInteger len = htons([data length]);

					[outBuffer release];
					outBuffer = [[NSMutableData dataWithCapacity:[data length] + 2] retain];
					[outBuffer appendBytes:&len length:2];
					[outBuffer appendData:data];
					outPos = 0;
				} else {
					// no more data to send
					break;
				}
			}

			maxLen = ([outBuffer length] - outPos) < 4096 ? ([outBuffer length] - outPos) : 4096;
			sentBytes = [outStream write:([outBuffer mutableBytes] + outPos) maxLength:maxLen];
			outPos += sentBytes;

			if ((outPos + 1) >= [outBuffer length]) {
				[outBuffer release];
				outBuffer = nil;
				outPos = 0;
			}
		} while (sentBytes >= maxLen);
	}
```


["play"]
["submit", {"word": "erlang"}]

Erlang:

```
idle({data, [<<"play">> | _]},
      StateData = #sd{player=Player}) ->

   %% handle command
```

Look how pretty this is!



20-30 lines of code to make it reasonable.


Game Logic

OTP has a whole behavior for this called gen_fsm.

OTP has a whole behavior for this called gen_fsm.

```
authing({data, [<<"auth">>, Props]}, State) ->

        %% authenticate


waiting({game_start, Game, Letters, Time, Players},
            State) ->

        %% send notification to client


playing({data, [<<"submit">>, Props]},
            StateData = #sd{game=Game}) ->

        %% handle word
```

OTP has a whole behavior for this called gen_fsm.

OTP again has a behavior for this kind of state bucket, gen_server

Receives:

Timer messages:
tick

Found words:
{submit_word, Word}

submit_word is an RPC call, since we must return the success value and the new scores

when it gets a tick, it checks if the game is over

when it gets a word, it has to verify it is valid and hasn't already been found

Sends:

{game_start, ...}

{clock, TimeLeft}

{word_found, Word, Who, ...}

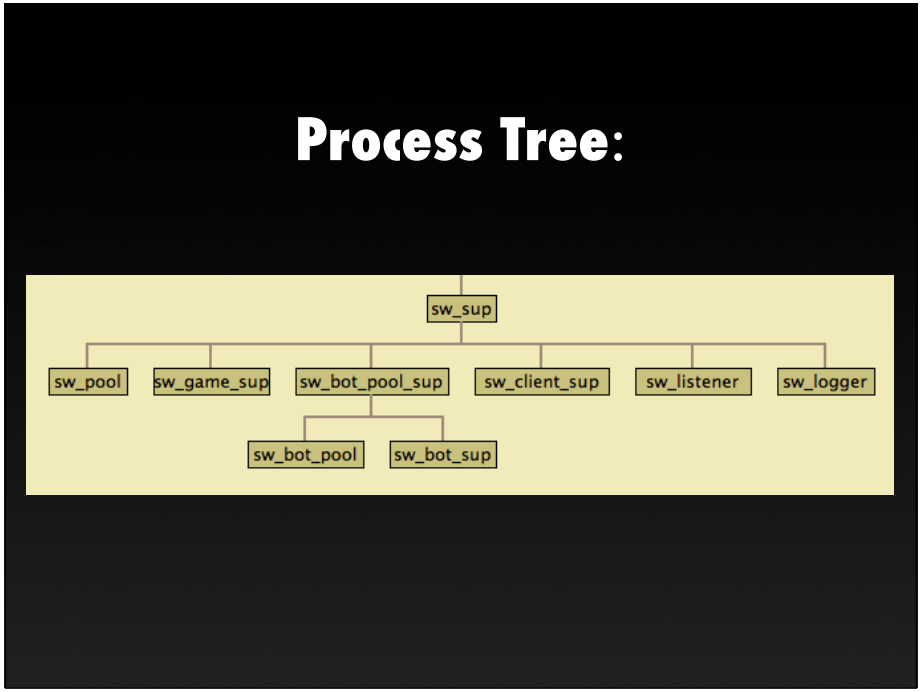
{game_end, Winner, ...}

Server Design

The server is an
OTP application.

The pieces:

clients
games
socket listener
waiting pool



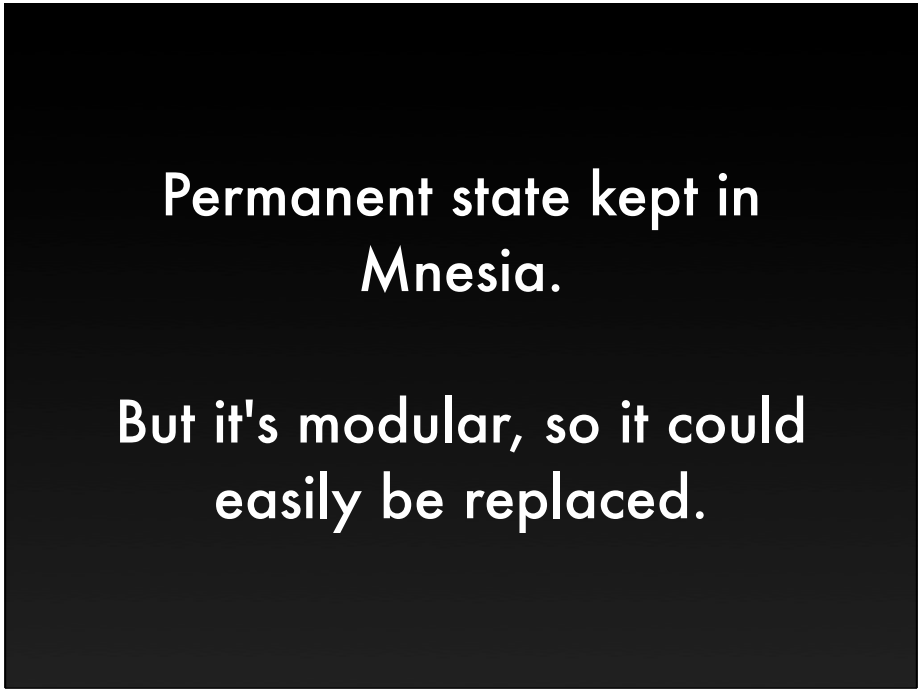
Each game is a process which will be linked to the game supervisor

Each client is a process linked to the client supervisor.

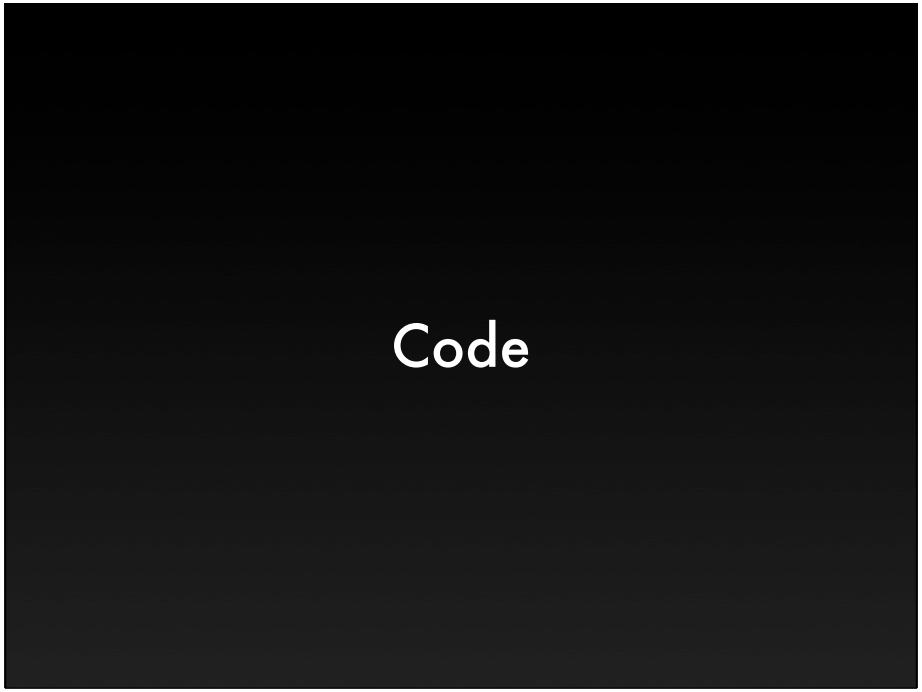
If a client crashes, the rest of the server is fine. If a game crashes, only the players are affected

One process per client.
One process per game.

1000 players
500 games
4 service processes
5 supervisors
= 1509 processes



There is a module called sw_db_mnesia, and the that is passed into the rest of the server. Changing the database means adjusting the configuration and creating a new sw_db_foo module that implements the same API.


Code

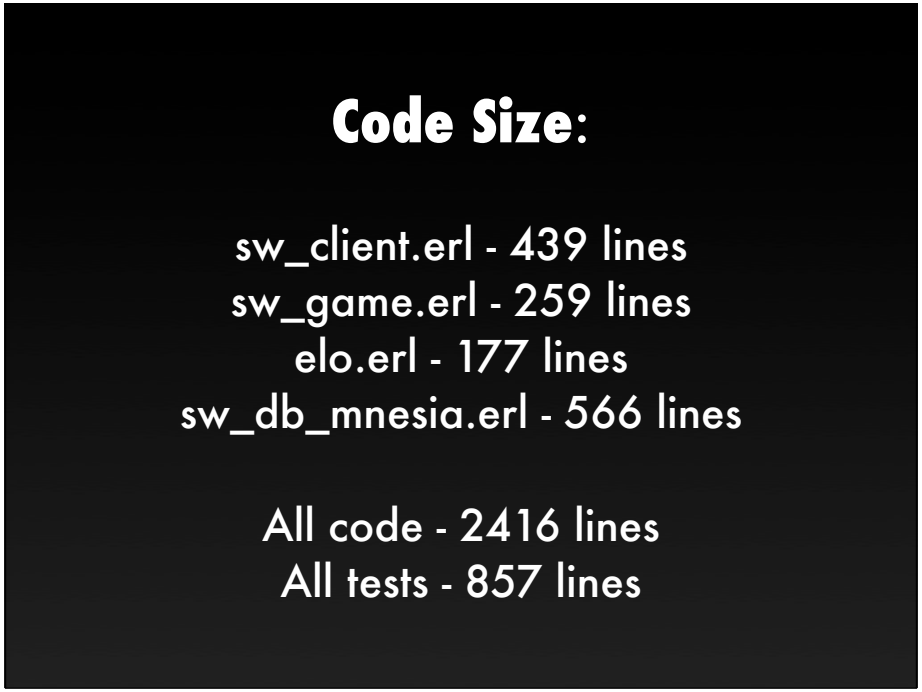
```
handle_info({tcp, Socket, Data}, StateName, StateData) ->
    %% set the socket so we can receive another data packet
    ok = inet:setopts(Socket, [{active, once}]),

    %% verify that Data is a JSON object
    case catch mochijson2:decode(Data) of
        {'EXIT', _} ->
            {stop, invalid_json, StateData};
        [_Cmd] = Json ->
            ?MODULE:StateName({data, Json}, StateData);
        [_Cmd, _Props] = Json ->
            ?MODULE:StateName({data, Json}, StateData);
        _ ->
            {stop, bad_request, StateData}
    end;
```

```
- (void)readAvailableData {
	// read all available data appending to inBuffer
	uint8_t buf[4096];
	NSUInteger len;
	NSMutableArray *packetQueue = [NSMutableArray arrayWithCapacity:5];

	do {
		len = [inStream read:buf maxLength:4096];
		[inBuffer appendBytes:buf length:len];

		while (1) {
			if (packetLength < 0 && [inBuffer length] >= 2) {
				// we have enough data to read the packet length
				packetLength = ntohs((uint16_t)(*(uint16_t *)[inBuffer mutableBytes]));
				inPos += 2;
			} else {
				break;
			}

			if (packetLength >= 0 && ([inBuffer length] - inPos) >= packetLength) {
				NSData *data = [NSData dataWithBytes:([inBuffer mutableBytes] + inPos) length:packetLength];
				NSString *json = [[[NSString alloc] initWithData:data encoding:NSUTF8StringEncoding] autorelease];

				NSLog(@"RECVING: %@", json);

				[packetQueue addObject:[json JSONValue]];

				[inBuffer replaceBytesInRange:NSMakeRange(0, 2 + packetLength) withBytes:NULL length:0];
				inPos = 0;
				packetLength = -1;
			} else {
				break;
			}
		}
	} while (len == 4096);

	for (id packet in packetQueue) {
		[self notifyPacket:packet];
	}
}
```

```
idle({data, [<<"play">> | _]}, StateData = #sd{player=Player}) ->
    case sw_pool:join_pool(Player) of
        {ok, joined} ->
            send([play_ok], StateData),
            {next_state, waiting, StateData, ?PING_INTERVAL};
        {error, _} ->
            send([play_fail, {struct, [{reason, <<"Internal error">>}]}],
                StateData),
            {next_state, idle, StateData, ?PING_INTERVAL}
    end;
```

DB Functions

```
-export([start/0,
        stop/1,
        get_player/3,
        create_player/2,
        create_player/5,
        save_player/2,
        move_player/3,
        authenticate/2,
        pick_puzzle/1,
        get_words/2,
        ...]).
```



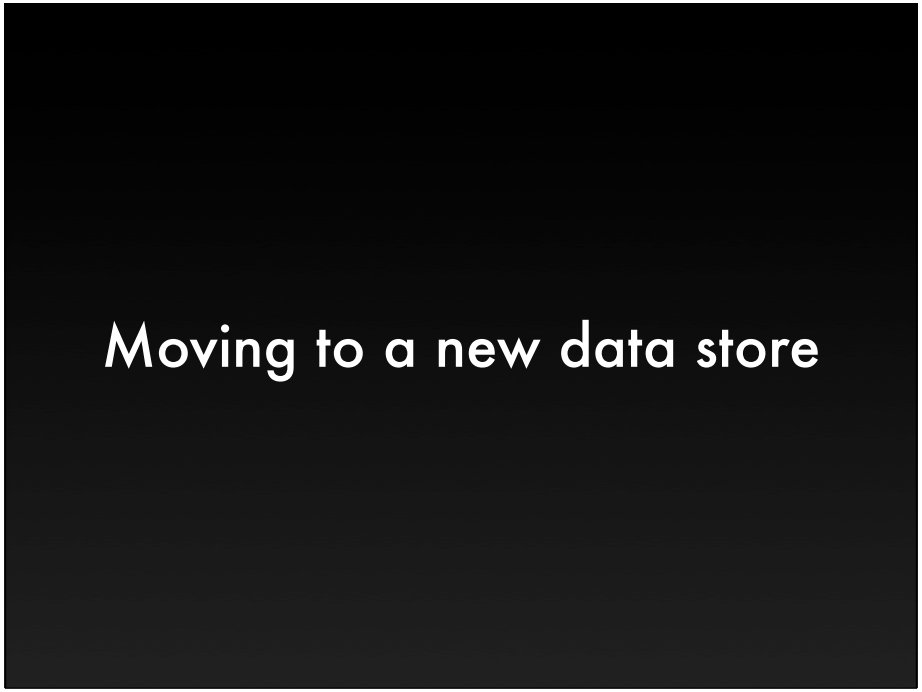
At Erlang Factory Lite LA I said I expected the whole project to come in under 2k lines. I was pretty close.

Performance:

Load tested 2000 clients on my laptop

Currently running on the cheapest Linode


The Future



Pretty easy. Just write a new module and change the applications configuration.

Main issue would be that you won't have easy access to Erlang terms; need to serialize/deserialize to SQL or JSON or something else.

Fire up another node and put both of them behind a load balancer and you're done.

The sw_pool process could exist only on one node or on multiple.

One of the design goals was to be extremely cheap to deploy, because there is no recurring revenue per customer.

Little custom protocols and network servers are right in Erlang's sweet spot.

It's extremely simple. It's really small. That makes is cheap and easy to maintain.

It's very cheap to deploy, which means I'm not going to lose money in infrastructure costs.

An equivalent system in Python might be similarly small, but future proofing would be significantly harder. Maintaining state in a real-time game server is A LOT harder; I've done it.

Not everything is in Erlang. I used Python for massaging the word list and creating the game databases. It can certainly be done in Erlang, but text file manipulations aren't really in the sweet spot.

jack@metajack.im
@metajack
http://metajack.im

http://snackwords.com